RANDOM COMBAT

PROJET DE LIFAPCD FAIT PAR Chahd Aloui et Jules Coasne

"RandomCombat" est un jeu basé sur des combats aléatoires entre un personnage joueur et différents ennemis. Le jeu utilise la bibliothèque SDL (Simple DirectMedia Layer) pour la gestion des graphismes et des événements. L'objectif du joueur est de survivre au plus de combats successifs possible, chaque combat devenant progressivement plus difficile.

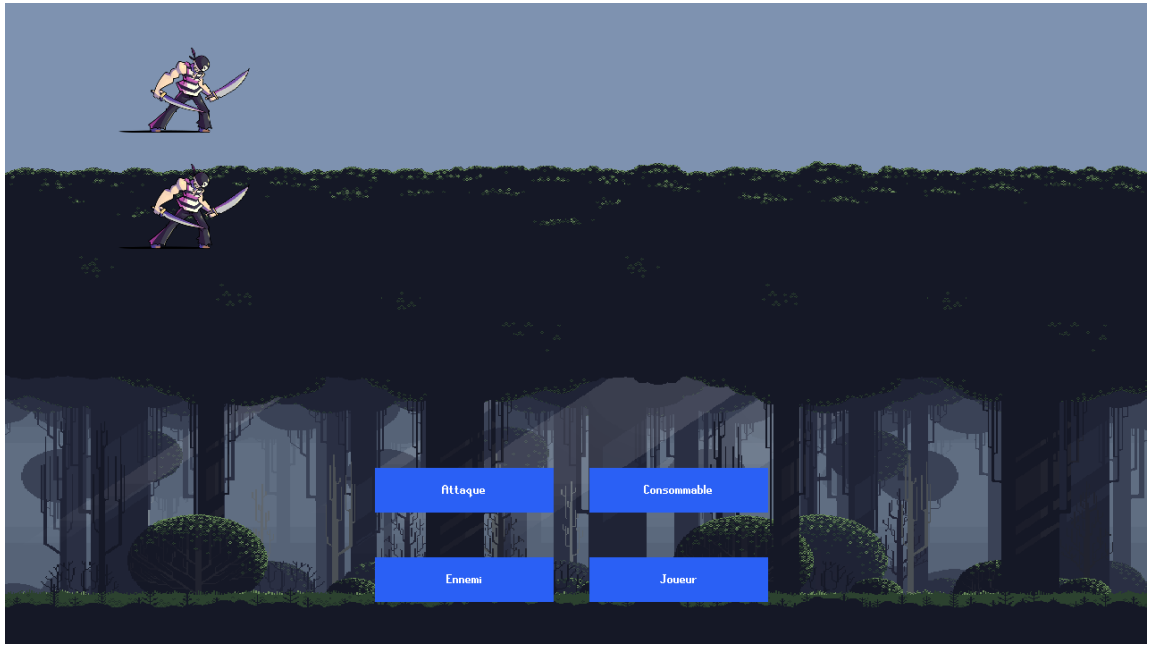
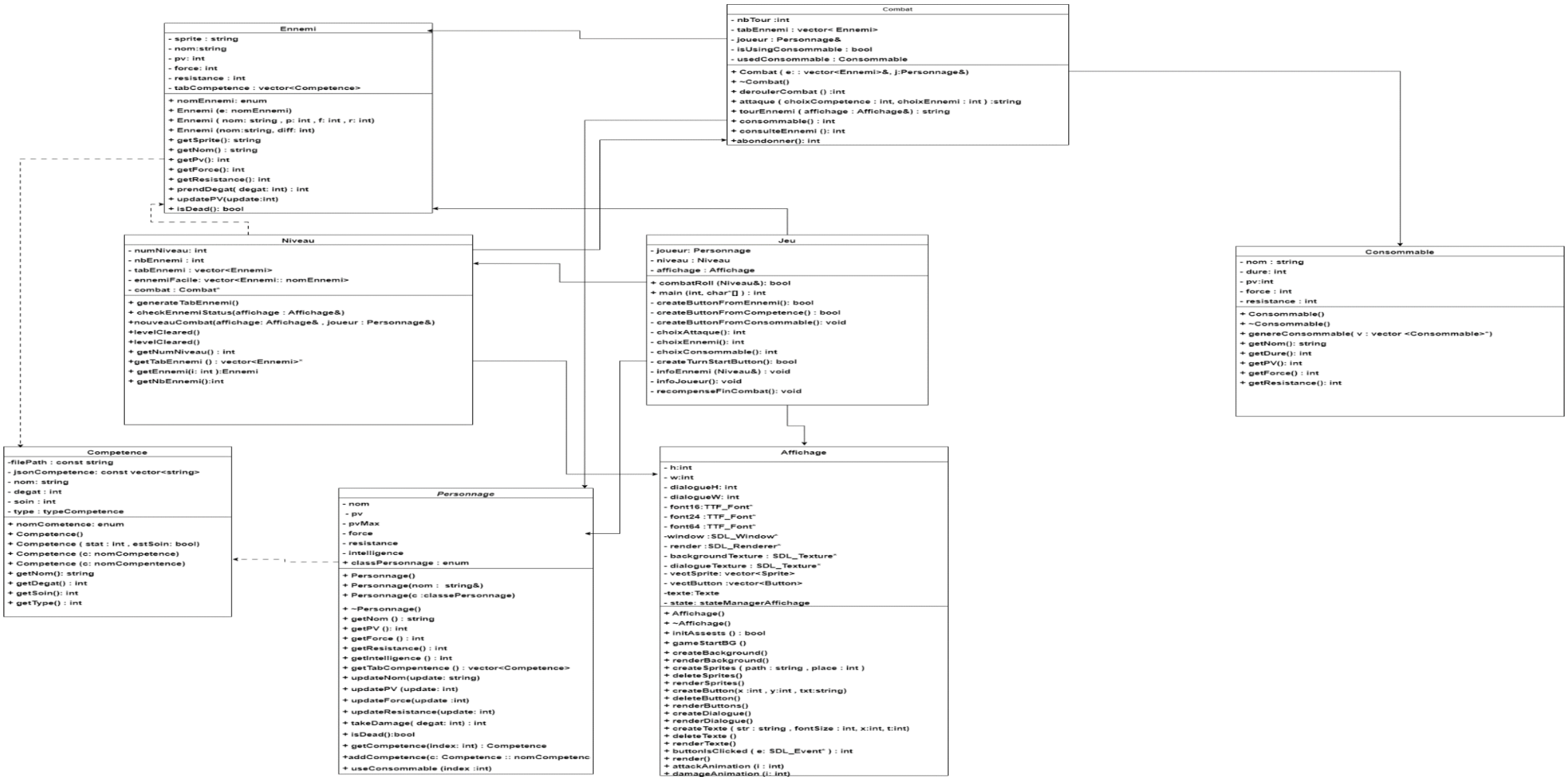


Diagramme des classes



1. Difficultés

Gérer plusieurs textures :

Dans notre code, nous avons utilisé des textures pour représenter différents éléments graphiques du jeu, tels que les ennemis, les boutons et les arrière-plans. La gestion de plusieurs textures nécessite de charger, de manipuler et de libérer correctement la mémoire des textures pour éviter les erreurs d'affichage. Il est important de gérer soigneusement les cycles de vie des textures et de s'assurer qu'elles sont chargées et déchargées de manière appropriée.

```
void Affichage::gameStartBG(){
    SDL_Surface * s = SDL_CreateRGBSurface(0, w, h, 32, 0, 0, 0, 0);
    if(s == NULL){
        cout << "Erreur lors de l'initialisation de la surface startBG" << SDL_GetError() << endl;
    }
    SDL_SetSurfaceBlendMode(s, SDL_BLENDMODE_BLEND);

    SDL_Texture * t = SDL_CreateTextureFromSurface(renderer, s);
    if(t == NULL){
        cout << "Erreur lors de l'initialisation de la texture startBG" << SDL_GetError() << endl;
    }


    SDL_RenderCopy(renderer, t, NULL, NULL);
    SDL_FreeSurface(s);
}

void Affichage::createBackground(){
    SDL_Surface *background_surface = IMG_Load("data/bg/background.png");
    if(background_surface == NULL){
        cout << "Erreur chargement du terrain" << SDL_GetError() << endl;
    }

    backgroundTexture = SDL_CreateTextureFromSurface(renderer, background_surface);
    SDL_FreeSurface(background_surface);
}
```

2. Difficultés

Les boutons :

Les boutons sont des éléments essentiels de l'interface utilisateur de notre jeu, permettant au joueur d'effectuer des actions telles que sélectionner une compétence ou naviguer dans les menus. La difficulté réside dans la création, le positionnement et l'interaction avec les boutons à l'écran. Il est nécessaire de détecter les clics de souris sur les boutons, de gérer les états des boutons et de déclencher les actions associées lorsque les boutons sont activés. Une bonne gestion des boutons garantit une expérience utilisateur fluide et intuitive.

```
void Affichage::renderButtons(){
    for(size_t i = 0; i < vectButton.size(); i++){
        SDL_Surface * surface = TTF_RenderText_Solid(font16, vectButton[i].texte, SDL_Color {255, 255, 255});
        SDL_Texture * texture = SDL_CreateTextureFromSurface(renderer, surface);

        SDL_Rect txtRect;
        txtRect.w = surface->w;
        txtRect.h = surface->h;
        txtRect.x = vectButton[i].descRect.x + (vectButton[i].descRect.w - txtRect.w) / 2;
        txtRect.y = vectButton[i].descRect.y + (vectButton[i].descRect.h - txtRect.h) / 2;

        SDL_RenderCopy(renderer, vectButton[i].texture, NULL, &vectButton[i].descRect);
        SDL_RenderCopy(renderer, texture, NULL, &txtRect);

        SDL_FreeSurface(surface);
    }
}
```

```
int Affichage::buttonIsClicked(SDL_Event * e){
    int x = e->motion.x;
    int y = e->motion.y;

    for(int i = 0; i < vectButton.size(); i++){
        SDL_Rect r = vectButton[i].descRect;

        if((x > r.x && x < r.x + r.w) && (y > r.y && y < r.y + r.h)){
            return i + 1;
        }
    }

    return 0;
}
```

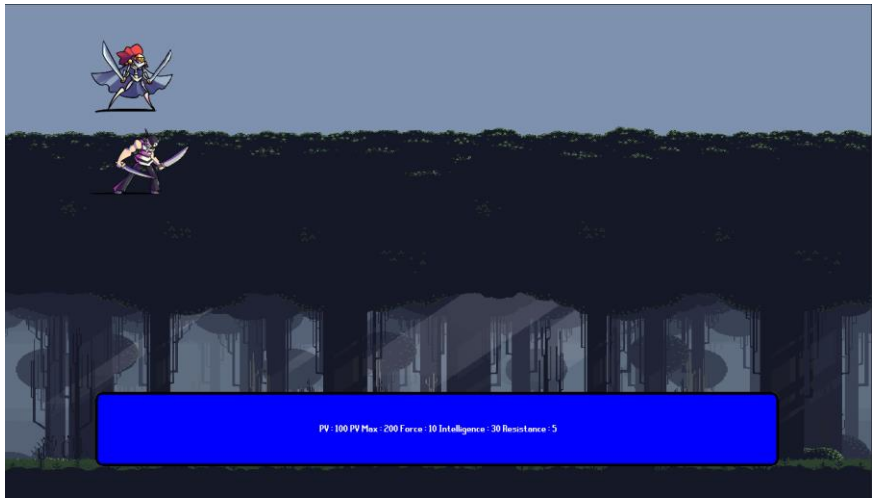
3. Classes interréssentes

Classe Ennemi :

Encapsulation des données : La classe Ennemi encapsule les données relatives à un ennemi spécifique, telles que son nom, ses points de vie, sa force, sa résistance, etc. Cela permet de regrouper toutes les informations concernant un ennemi en un seul objet, facilitant ainsi la manipulation et la gestion des ennemis dans votre jeu.

Flexibilité : Grâce aux différents constructeurs de la classe Ennemi, nous avons créer des ennemis de manière flexible en chargeant leurs données à partir de fichiers JSON, en spécifiant directement leurs attributs ou en les ajustant en fonction d'un niveau de difficulté. Cela offre une grande flexibilité dans la conception et la création d'ennemis variés pour votre jeu.

Modularité : La classe Ennemi est conçue de manière modulaire, avec des méthodes telles que prendDegat() pour gérer les dégâts reçus par l'ennemi et isDead() pour vérifier si l'ennemi est mort. Cette modularité facilite l'extension et la maintenance du code, permettant d'ajouter facilement de nouvelles fonctionnalités ou de modifier le comportement des ennemis au besoin.



4. Classes intéressantes

Classe Personnage :

Choix du personnage : Le joueur à la possibilité de choisir en début de partie son personnage, la classe se chargent de créer le personnage avec les bonnes statistiques et compétences

Gestion des compétences : La classe Personnage gère les compétences du personnage jouable à travers le vector tabCompetence. Cela permet d'ajouter, de supprimer ou de modifier facilement les compétences du personnage, offrant ainsi une grande variété de styles de jeu et de stratégies possibles.

Inventaire de consommables : La classe Personnage gère également les consommables du personnage à travers le vector tabConsommable. Cela permet aux joueurs d'accumuler et d'utiliser des objets de soin ou des boosts pendant le jeu, ce qui ajoute une dimension stratégique supplémentaire à l'expérience de jeu.

Conclusion